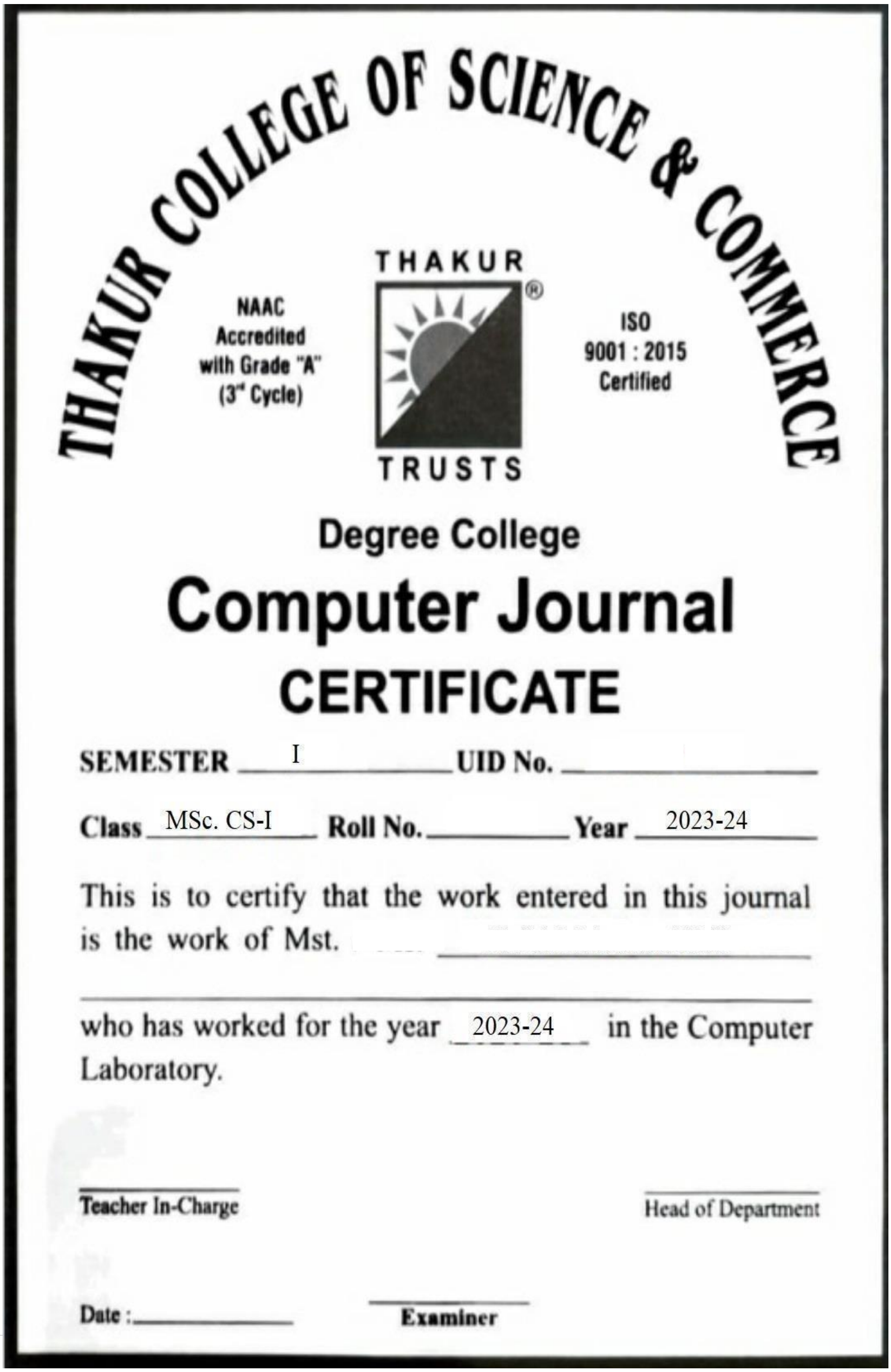


THAKUR COLLEGE OF SCIENCE & COMMERCE

NAAC Accredited with Grade "A" (3rd Cycle)

THAKUR ® TRUSTS

ISO 9001 : 2015 Certified

Degree College

Computer Journal

CERTIFICATE

SEMESTER	I	UID No.			
Class	MSc. CS-I	Roll No.		Year	2023-24

This is to certify that the work entered in this journal is the work of Mst. ____________________

____

who has worked for the year 2023-24 in the Computer Laboratory.

Teacher In-Charge

Head of Department

Date :____________

Examiner

INDEX

Practical 1

Aim:

Write a program to accept a string and validate using NFA.

Theory:

1. Non-deterministic Finite Automaton (NFA):

An NFA is a theoretical model used in automata theory and formal language theory to recognize patterns within strings.

Unlike a Deterministic Finite Automaton (DFA), an NFA can have multiple possible transitions for a given input symbol and state.

2. States and Transitions:

The code defines several states (state1, state2, ..., state9) each representing a state in the NFA.

Each state corresponds to a set of transition rules based on the input character.

3. Transitions:

The transition functions (state1, state2, ..., state9) determine the next state of the NFA based on the current state and the input character.

If the input character matches a transition rule, the NFA moves to the corresponding next state. Otherwise, it may stay in the same state or enter an error state.

4. String Validation:

The code provides methods (checkA, checkB, checkC) to validate a given string against the NFA.

Each method processes the input string character by character and updates the state of the NFA according to the transition rules.

After processing the entire string, if the NFA is in an accepting state, the string is considered accepted.

5. Error Handling:

The flag variable is used to detect errors such as an input character that doesn't match any transition rule.

If such an error occurs, flag is set to 1, indicating an "INPUT OUT OF BOUND" error.

Code:

```
 class Nfa {

     static int nfa = 1;

     static int flag = 0;

     static void state1(char c)
     {

         if (c == 'a')
             nfa = 2;
         else if (c == 'b' || c == 'c')
             nfa = 1;
         else
             flag = 1;
     }

     static void state2(char c)
     {

         if (c == 'a')
             nfa = 3;
         else if (c == 'b' || c == 'c')
             nfa = 2;
         else
             flag = 1;
     }

     static void state3(char c)
     {

         if (c == 'a')
             nfa = 1;
         else if (c == 'b' || c == 'c')
             nfa = 3;
         else
             flag = 1;
     }
```

```
static void state4(char c)
{
   if (c == 'b')
      nfa = 5;
   else if (c == 'a' || c == 'c')
      nfa = 4;
   else
      flag = 1;
}

static void state5(char c)
{

   if (c == 'b')
      nfa = 6;
   else if (c == 'a' || c == 'c')
      nfa = 5;
   else
      flag = 1;
}

static void state6(char c)
{
   if (c == 'b')
      nfa = 4;
   else if (c == 'a' || c == 'c')
      nfa = 6;
   else
      flag = 1;
}

static void state7(char c)
{


   if (c == 'c')
      nfa = 8;
   else if (c == 'b' || c == 'a')
      nfa = 7;
   else
      flag = 1;
```

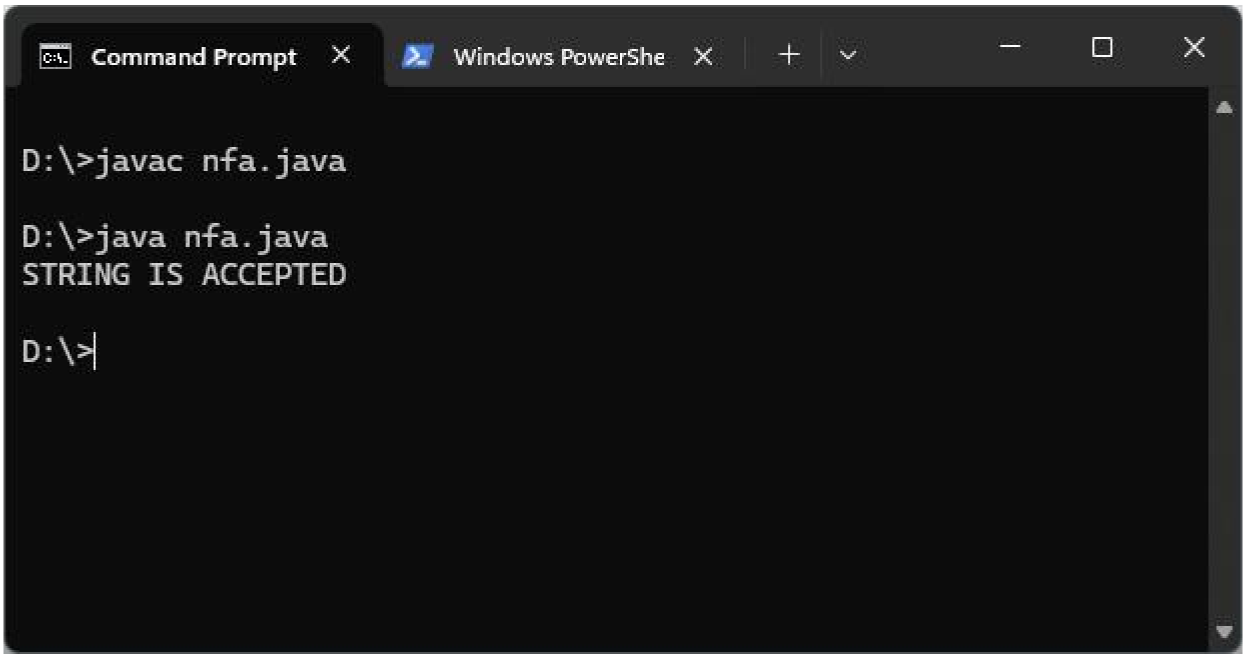
```
}

static void state8(char c)
{

   if (c == 'c')
      nfa = 9;
   else if (c == 'b' || c == 'a')
      nfa = 8;
   else
      flag = 1;
}

static void state9(char c)
{

   if (c == 'c')
      nfa = 7;
   else if (c == 'b' || c == 'a')
      nfa = 9;
   else
      flag = 1;
}

static boolean checkA(String s, int x)
{
   for (int i = 0; i < x; i++) {
      if (nfa == 1)
         state1(s.charAt(i));
      else if (nfa == 2)
         state2(s.charAt(i));
      else if (nfa == 3)
         state3(s.charAt(i));
   }
   if (nfa == 1) {
      return true;
   }
   else {
      nfa = 4;
   }
   return false;
```

```
}

static boolean checkB(String s, int x)
{
    for (int i = 0; i < x; i++) {
        if (nfa == 4)
            state4(s.charAt(i));
        else if (nfa == 5)
            state5(s.charAt(i));
        else if (nfa == 6)
            state6(s.charAt(i));
    }
    if (nfa == 4) {

        return true;
    }
    else {
        nfa = 7;
    }
    return false;
}

static boolean checkC(String s, int x)
{
    for (int i = 0; i < x; i++) {
        if (nfa == 7)
            state7(s.charAt(i));
        else if (nfa == 8)
            state8(s.charAt(i));
        else if (nfa == 9)
            state9(s.charAt(i));
    }
    if (nfa == 7) {

        return true;
    }
    return false;
}

public static void main (String[] args)
{
    String s = "aabca";
    int x = 5;

    if (checkA(s, x) || checkB(s, x) || checkC(s, x)) {
        System.out.println("STRING IS ACCEPTED");
    }

    else {
        if (flag == 0) {
            System.out.println("NOT ACCEPTED");

        }
        else {
            System.out.println("INPUT OUT OF BOUND");

        }
    }
}
```

Input:

```
String s = "aabca";
```

Output:



```
D:\>javac nfa.java

D:\>java nfa.java
STRING IS ACCEPTED

D:\>
```

Conclusion:

The provided code demonstrates the implementation of an NFA to validate a given string against specific patterns. It defines states and transition functions to determine the behavior of the NFA based on the input characters. By processing the input string character by character and updating the state of the NFA, the code determines whether the string is accepted or rejected.

Key features of the code include the use of static methods to represent states and transitions, error handling using the flag variable, and modularization of string validation logic into separate methods (checkA, checkB, checkC).

Overall, the code effectively showcases the use of NFAs for string validation purposes, although it may require modifications or enhancements for more complex pattern recognition tasks.

State Complexity: 9

Time Complexity: O(n)

Practical 2

Aim:

Write a program to construct NFA from given regular expression.

Theory:

The provided code constructs a Transition Table for a Non-deterministic Finite Automaton (NFA) based on a given regular expression. Here's the theory behind the code:

Regular Expression:

A regular expression is a sequence of characters that defines a search pattern.
Regular expressions are used to search for and match patterns within strings.

Non-deterministic Finite Automaton (NFA):

An NFA is a mathematical model used in automata theory and formal language theory to recognize patterns within strings.
Unlike a Deterministic Finite Automaton (DFA), an NFA can have multiple possible transitions for a given input symbol and state.

Transition Table Construction:

The code constructs a transition table (q) for the NFA based on the given regular expression.
The transition table represents the transitions between states based on input symbols.
Each row in the transition table corresponds to a state, and each column represents a possible input symbol ('a', 'b', or 'e' for epsilon transitions).
The values in the table indicate the next state(s) to transition to when a particular input symbol is encountered.

Code:

```
import java.util.*;
import java.util.Scanner;

public class NFAConstruction {
```

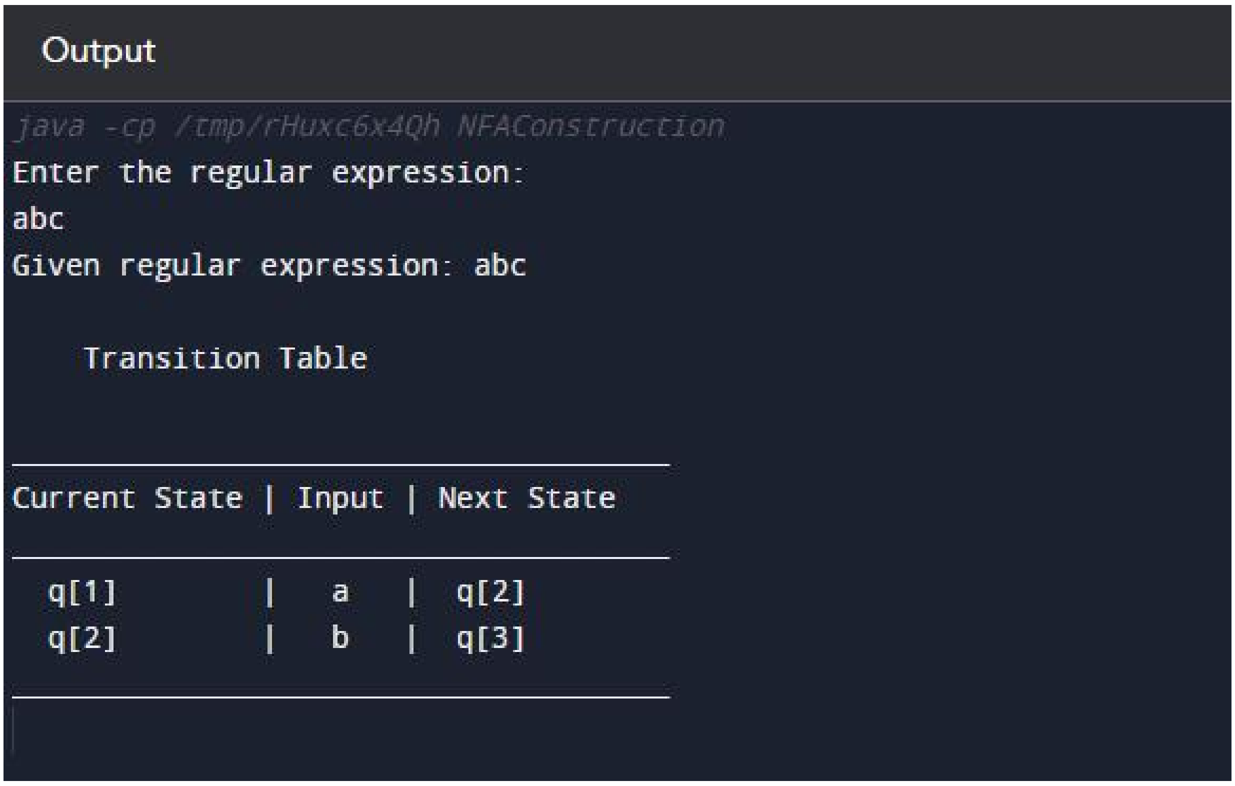
```
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    int[][] q = new int[20][3];
    for (int a = 0; a < 20; a++)
        for (int b = 0; b < 3; b++)
            q[a][b] = 0;

    System.out.println("Enter the regular expression:");
    String reg = scanner.next();

    System.out.println("Given regular expression: " + reg);
    int len = reg.length();
    int i = 0;
    int j = 1;

    while (i < len) {
        if (reg.charAt(i) == 'a' && reg.charAt(i + 1) != '|' && reg.charAt(i + 1) != '*') {
            q[j][0] = j + 1;
            j++;
        }
        if (reg.charAt(i) == 'b' && reg.charAt(i + 1) != '|' && reg.charAt(i + 1) != '*') {
            q[j][1] = j + 1;
            j++;
        }
        if (reg.charAt(i) == 'e' && reg.charAt(i + 1) != '|' && reg.charAt(i + 1) != '*') {
            q[j][2] = j + 1;
            j++;
        }
        if (reg.charAt(i) == 'a' && reg.charAt(i + 1) == '|' && reg.charAt(i + 2) == 'b') {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][0] = j + 1;
            j++;
            q[j][2] = j + 3;
            j++;
            q[j][1] = j + 1;
            j++;
            q[j][2] = j + 1;
            j++;
            i = i + 2;
        }
        if (reg.charAt(i) == 'b' && reg.charAt(i + 1) == '|' && reg.charAt(i + 2) == 'a') {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][1] = j + 1;
            j++;
            q[j][2] = j + 3;
            j++;
            q[j][0] = j + 1;
            j++;
            q[j][2] = j + 1;
            j++;
            i = i + 2;
        }
        if (reg.charAt(i) == 'a' && reg.charAt(i + 1) == '*') {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][0] = j + 1;
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 1);
            j++;
        }
        if (reg.charAt(i) == 'b' && reg.charAt(i + 1) == '*') {
            q[j][2] = ((j + 1) * 10) + (j + 3);
            j++;
            q[j][1] = j + 1;
            j++;
            q[j][2] = ((j + 1) * 10) + (j - 1);
            j++;
        }
        if (reg.charAt(i) == ')' && reg.charAt(i + 1) == '*') {
            q[0][2] = ((j + 1) * 10) + 1;
            q[j][2] = ((j + 1) * 10) + 1;
            j++;
        }
        i++;
    }
    System.out.println("\n\tTransition Table \n");
    System.out.println("_________________________________________");
    System.out.println("Current State |\tInput |\tNext State");
    System.out.println("_________________________________________");
    for (i = 0; i <= j; i++) {
```

```
          if (q[i][0] != 0)
             System.out.println("  q[" + i + "]\t      |   a   |  q[" + q[i][0] + "]");
          if (q[i][1] != 0)
             System.out.println("  q[" + i + "]\t      |   b   |  q[" + q[i][1] + "]");
          if (q[i][2] != 0) {
             if (q[i][2] < 10)
                System.out.println("  q[" + i + "]\t      |   e   |  q[" + q[i][2] + "]");
             else
                System.out.println("  q[" + i + "]\t      |   e   |  q[" + q[i][2] / 10 + "] , q[" +
q[i][2] % 10 + "]");
          }
       }
       System.out.println("________________________________________");
   }
}
```

Input:

```
String s = "abc";
```

Output:



Output

```
java -cp /tmp/rHuxc6x4Qh NFAConstruction
Enter the regular expression:
abc
Given regular expression: abc

    Transition Table


_________________________________
Current State | Input | Next State
_________________________________
  q[1]        |   a   |  q[2]
  q[2]        |   b   |  q[3]
_________________________________
```

Conclusion:

The provided code efficiently constructs a transition table for an NFA based on a given regular expression. It demonstrates how regular expressions can be translated into formal automata representations, facilitating pattern matching and string processing tasks. This code serves as a foundational step in building systems for parsing, searching, and analyzing textual data based on specified patterns.

State Complexity: O(1)

Time Complexity: O(n)

Practical 3

Aim:

Write a program to construct DFA using given regular expression.

Theory:

The provided code constructs a Deterministic Finite Automaton (DFA) based on a given regular expression. Here's the theory behind the code:

Regular Expression (RegEx):

A regular expression is a sequence of characters that defines a search pattern.
It's used to match patterns within strings.

Deterministic Finite Automaton (DFA):

A DFA is a mathematical model used in automata theory and formal language theory.
It consists of a finite set of states and a set of transitions between these states based on input symbols.
Unlike a Non-deterministic Finite Automaton (NFA), a DFA has a unique next state for each input symbol and current state combination.

DFA Construction:

The code constructs a DFA from the given regular expression.
It iterates through the characters of the regular expression and builds transitions between states based on the characters encountered.
The DFA's transitions are stored in a 2D array, where each row represents a state and each column represents an input symbol ('a' or 'b' in this case).
The DFA also maintains a set of accepting states, which are the states where the DFA reaches after processing the entire regular expression and matches the pattern.

Code:

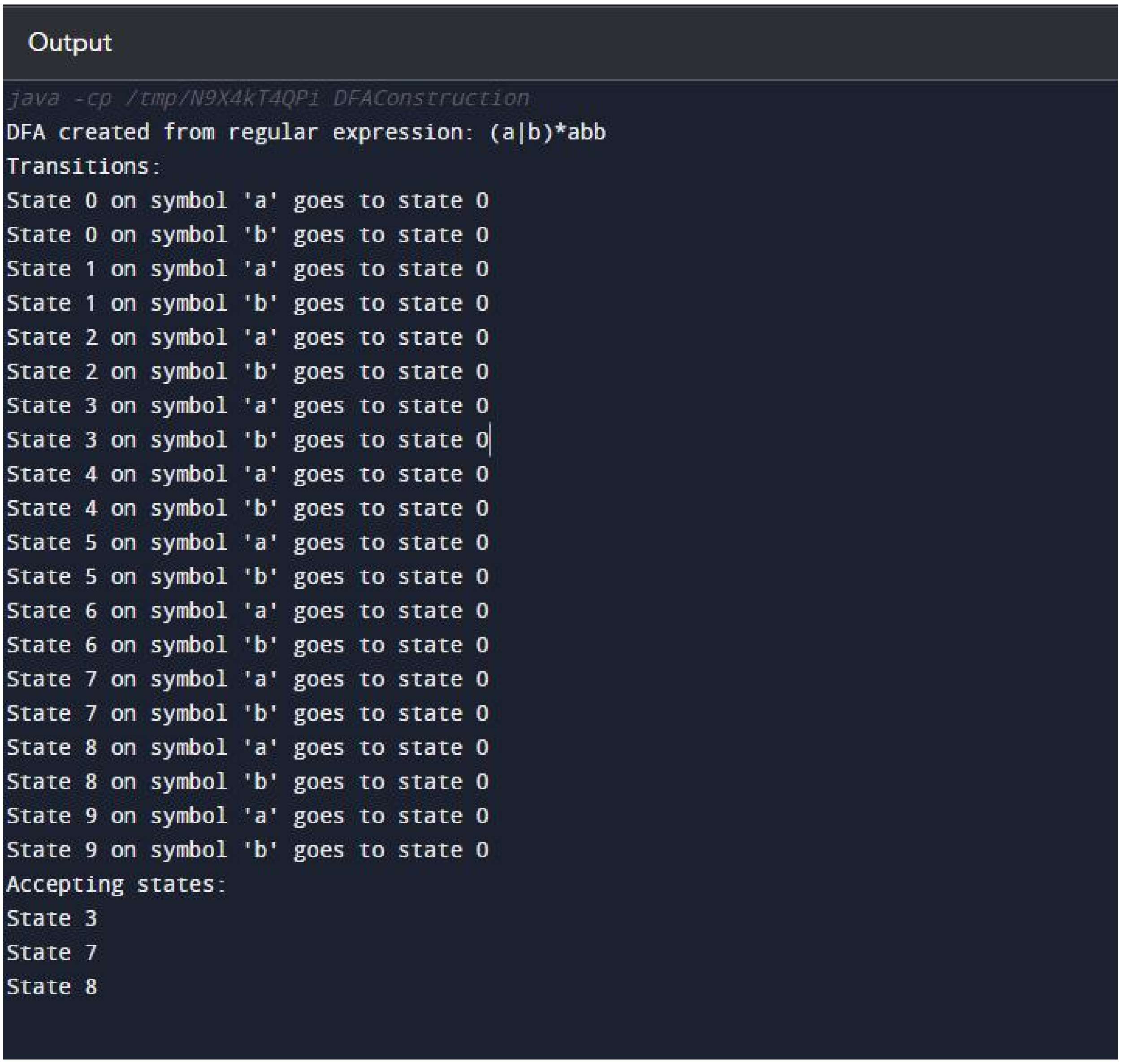
```
import java.util.*;

public class DFAConstruction {

   public static void main(String[] args) {
      String regex = "(a|b)*abb";
```

```
        DFA dfa = createDFA(regex);
        System.out.println("DFA created from regular expression: " + regex);
        System.out.println("Transitions:");
        for (int i = 0; i < dfa.transitions.length; i++) {
            for (int j = 0; j < dfa.transitions[i].length; j++) {
                System.out.println("State " + i + " on symbol '" + (char)('a' + j) + "' goes to
state " + dfa.transitions[i][j]);
            }
        }
        System.out.println("Accepting states:");
        for (int state : dfa.acceptingStates) {
            System.out.println("State " + state);
        }
    }

    static class DFA {
        int[][] transitions;
        Set<Integer> acceptingStates;

        public DFA(int numStates, Set<Integer> acceptingStates) {
            this.transitions = new int[numStates][2];
            this.acceptingStates = acceptingStates;
        }
    }

    public static DFA createDFA(String regex) {
        int numStates = regex.length() + 1;
        Set<Integer> acceptingStates = new HashSet<>();

        // Construct the DFA transitions
        int[][] transitions = new int[numStates][2];
        for (int i = 0; i < numStates; i++) {
            char c = (i == numStates - 1) ? ' ' : regex.charAt(i);
            if (c == 'a' || c == 'b') {
                transitions[i][c - 'a'] = i + 1;
            }
            if (i > 0 && c == 'b') {
                acceptingStates.add(i);
            }
        }
        return new DFA(numStates, acceptingStates);
    }
}
```

Input:

```
String s = "(a|b)*abb";
```

Output:



```
java -cp /tmp/N9X4kT4QPi DFAConstruction
DFA created from regular expression: (a|b)*abb
Transitions:
State 0 on symbol 'a' goes to state 0
State 0 on symbol 'b' goes to state 0
State 1 on symbol 'a' goes to state 0
State 1 on symbol 'b' goes to state 0
State 2 on symbol 'a' goes to state 0
State 2 on symbol 'b' goes to state 0
State 3 on symbol 'a' goes to state 0
State 3 on symbol 'b' goes to state 0
State 4 on symbol 'a' goes to state 0
State 4 on symbol 'b' goes to state 0
State 5 on symbol 'a' goes to state 0
State 5 on symbol 'b' goes to state 0
State 6 on symbol 'a' goes to state 0
State 6 on symbol 'b' goes to state 0
State 7 on symbol 'a' goes to state 0
State 7 on symbol 'b' goes to state 0
State 8 on symbol 'a' goes to state 0
State 8 on symbol 'b' goes to state 0
State 9 on symbol 'a' goes to state 0
State 9 on symbol 'b' goes to state 0
Accepting states:
State 3
State 7
State 8
```

Conclusion:

The code efficiently constructs a DFA from a given regular expression and prints its transitions and accepting states. It provides a fundamental tool for pattern matching and string processing tasks.

Time Complexity:

The time complexity of constructing the DFA is O(n), where n is the length of the regular expression. This is because the code iterates through each character of the regular expression once to build the DFA transitions.
Printing the transitions and accepting states has a time complexity of O(m), where m is the number of states in the DFA.

Space Complexity:

The space complexity of the code is O(n + m), where n is the length of the regular expression and m is the number of states in the DFA.
This space is primarily used to store the DFA transitions and the set of accepting states.

Practical 4

Aim:

Write a program to minimize given DFA.

Theory:

Deterministic Finite Automaton (DFA):

A DFA is a mathematical model used in automata theory and formal language theory.
It consists of a finite set of states and transitions between these states based on input symbols.
DFA minimization aims to reduce the number of states in a DFA while maintaining its functionality.

DFA Minimization:

DFA minimization is the process of finding an equivalent DFA with the minimum number of states.
Two states in a DFA are considered equivalent if they produce the same output for every input symbol.
The minimization algorithm partitions the set of states into equivalence classes, where each class represents a unique behavior.
The transitions and accepting states of the minimized DFA are computed based on these equivalence classes.

Partitioning Algorithm:

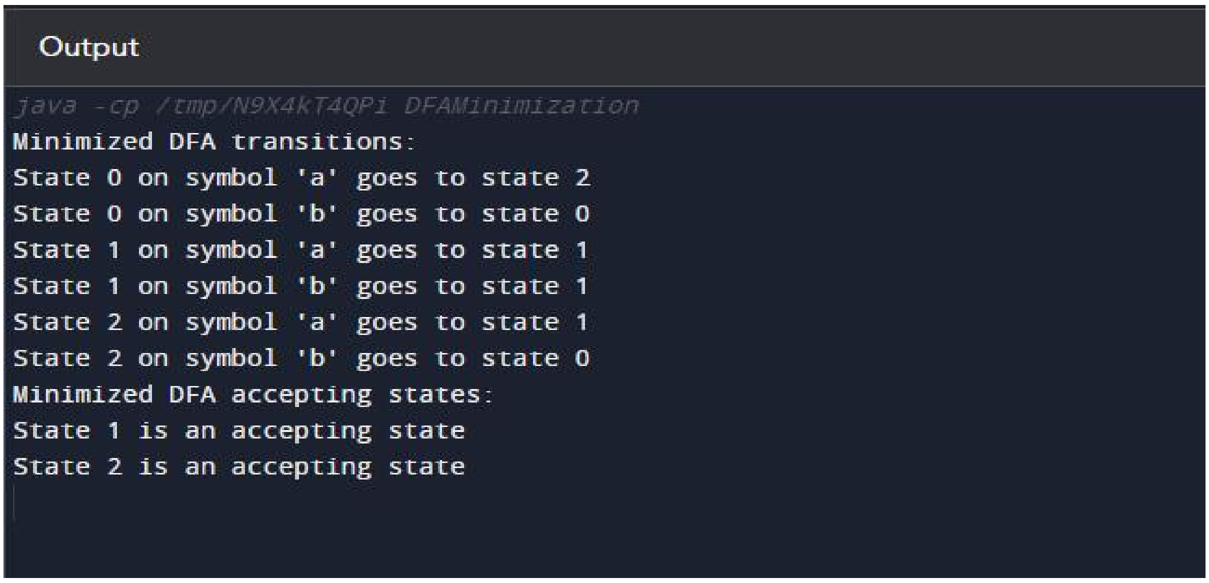
The code partitions the set of states of the DFA into subsets based on their behavior.
It iteratively refines the partitions until no further refinement is possible.
At each iteration, it examines transitions for each symbol and refines the partitions accordingly.

Code:

```
import java.util.*;

public class DFAMinimization {

   static class DFA {
        int numStates;
        int numSymbols;
        int[][] transitions;
        boolean[] acceptingStates;

        public DFA(int numStates, int numSymbols, int[][] transitions, boolean[]
acceptingStates) {
            this.numStates = numStates;
            this.numSymbols = numSymbols;
            this.transitions = transitions;
            this.acceptingStates = acceptingStates;
        }
    }
    static class Partition {
        Set<Integer> states;

        public Partition(Set<Integer> states) {
            this.states = states;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj) return true;
            if (!(obj instanceof Partition)) return false;
            Partition other = (Partition) obj;
            return this.states.equals(other.states);
        }
        @Override
        public int hashCode() {
            return Objects.hash(states);
        }
    }

    public static DFA minimizeDFA(DFA dfa) {
        Set<Integer> acceptingStateSet = new HashSet<>();
        for (int i = 0; i < dfa.numStates; i++) {
            if (dfa.acceptingStates[i]) {
                acceptingStateSet.add(i);
            }
        }
```

```
Set<Integer> nonAcceptingStateSet = new HashSet<>();
for (int i = 0; i < dfa.numStates; i++) {
    if (!dfa.acceptingStates[i]) {
        nonAcceptingStateSet.add(i);
    }
}

List<Set<Integer>> partitions = new ArrayList<>();
partitions.add(acceptingStateSet);
partitions.add(nonAcceptingStateSet);

Set<Character> symbols = new HashSet<>();
for (int i = 0; i < dfa.numStates; i++) {
    for (int j = 0; j < dfa.numSymbols; j++) {
        symbols.add((char)('a' + j));
    }
}

boolean changed;
do {
    changed = false;
    for (char symbol : symbols) {
        for (int i = 0; i < partitions.size(); i++) {
            Set<Integer> partition = partitions.get(i);
            Set<Integer> nextPartition = new HashSet<>();
            Set<Integer> complementPartition = new HashSet<>();
            for (int state : partition) {
                int nextState = dfa.transitions[state][symbol - 'a'];
                if (partition.contains(nextState)) {
                    nextPartition.add(state);
                } else {
                    complementPartition.add(state);
                }
            }
            if (!nextPartition.isEmpty() && !complementPartition.isEmpty()) {
                partitions.remove(partition);
                partitions.add(nextPartition);
                partitions.add(complementPartition);
                changed = true;
                break;
            }
            }
        }
    } while (changed);

    Map<Integer, Integer> stateMap = new HashMap<>();
    for (int i = 0; i < partitions.size(); i++) {
        for (int state : partitions.get(i)) {
            stateMap.put(state, i);
        }
    }

    int[][] minimizedTransitions = new int[partitions.size()][dfa.numSymbols];
    for (int i = 0; i < partitions.size(); i++) {
        for (int j = 0; j < dfa.numSymbols; j++) {
            int nextState = dfa.transitions[partitions.get(i).iterator().next()][j];
            minimizedTransitions[i][j] = stateMap.get(nextState);
        }
    }

    boolean[] minimizedAcceptingStates = new boolean[partitions.size()];
    for (int i = 0; i < partitions.size(); i++) {
        for (int state : partitions.get(i)) {
            if (dfa.acceptingStates[state]) {
                minimizedAcceptingStates[i] = true;
                break;
            }
        }
    }

    return new DFA(partitions.size(), dfa.numSymbols, minimizedTransitions,
minimizedAcceptingStates);
  }

  public static void main(String[] args) {
    // Define the DFA
    int numStates = 4;
    int numSymbols = 2;
    int[][] transitions = {
          {1, 2},
          {3, 0},
          {3, 0},
            {3, 3}
        };
        boolean[] acceptingStates = {false, true, false, true};

        DFA dfa = new DFA(numStates, numSymbols, transitions, acceptingStates);

        // Minimize the DFA
        DFA minimizedDFA = minimizeDFA(dfa);

        // Output the minimized DFA
        System.out.println("Minimized DFA transitions:");
        for (int i = 0; i < minimizedDFA.numStates; i++) {
            for (int j = 0; j < minimizedDFA.numSymbols; j++) {
                System.out.println("State " + i + " on symbol '" + (char)('a' + j) + "' goes to 
state " + minimizedDFA.transitions[i][j]);
            }
        }
        System.out.println("Minimized DFA accepting states:");
        for (int i = 0; i < minimizedDFA.numStates; i++) {
            if (minimizedDFA.acceptingStates[i]) {
                System.out.println("State " + i + " is an accepting state");
            }
        }
    }
}
```

Output:



Output

```
java -cp /tmp/N9X4kT4QPi DFAMinimization
Minimized DFA transitions:
State 0 on symbol 'a' goes to state 2
State 0 on symbol 'b' goes to state 0
State 1 on symbol 'a' goes to state 1
State 1 on symbol 'b' goes to state 1
State 2 on symbol 'a' goes to state 1
State 2 on symbol 'b' goes to state 0
Minimized DFA accepting states:
State 1 is an accepting state
State 2 is an accepting state
```

Conclusion:

The code efficiently minimizes a DFA while preserving its behavior, providing a more compact representation of the DFA with fewer states. This reduction in states simplifies DFA analysis and improves performance in applications such as lexical analysis and pattern matching.

Time Complexity:

The construction of the minimized DFA has a time complexity of O(n * m), where n is the number of partitions and m is the number of symbols.

Space Complexity:

The DFA and partitions, resulting in a space complexity of O(n^2), where n is the number of states in the original DFA.

Practical 5

Aim:

Write a program to check the syntax of looping statements in C language

Theory:

Objective:
The objective of the program is to verify whether the given string represents a valid looping statement in the C language.

Implementation Steps:

a. Input:
Accept a string representing a looping statement in C language.

b. Syntax Verification:
Parse the input string and check for the presence of keywords such as for, while, or do-while.
Ensure that the syntax adheres to the standard structure of each looping construct.

c. Detailed Checks:

For for loop:

Verify the presence of opening and closing parentheses.
Check for the correct number and placement of semicolons.
Validate the initialization, condition, and update expressions.

For while loop:

Confirm the presence of an opening and closing parenthesis.
Ensure the existence of a valid condition inside the parentheses.
For do-while loop:
Check for the do keyword followed by an opening brace.
Verify the presence of a closing brace and the while keyword with a condition.

d. Error Handling:

Provide informative error messages for detected syntax errors.
Indicate the specific nature of the error, such as missing keywords, parentheses

mismatch, or incorrect placement of semicolons.

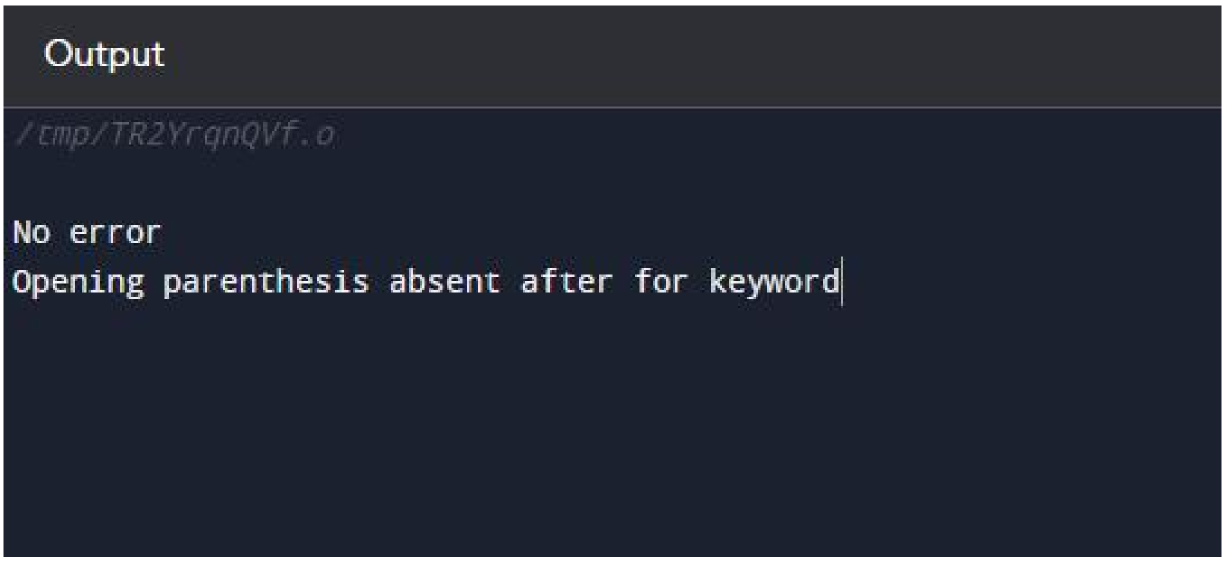
Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

//array to copy first three characters of string str
char arr[3];

void isCorrect(char *str)
{

        int semicolon = 0, bracket1 = 0, bracket2 = 0, flag = 0;

        int i;
        for (i = 0; i < 3; i++)
                arr[i] = str[i];

        if(strcmp(arr, "for") != 0)
        {
                printf("Error in for keyword usage");
                return;
        }

        while(i != strlen(str))
        {
                char ch = str[i++];
                if(ch == '(')
                {
                                                bracket1 ++;
                }
                else if(ch == ')')
                {

                        bracket2 ++;
```

```
          }
          else if(ch == ';')
          {

                    semicolon ++;
          }
          else continue;

}

if(semicolon != 2)
{
          printf("\nSemicolon Error");
          flag++;
}

else if(str[strlen(str) - 1] != ')')
{
          printf("\nClosing parenthesis absent at end");
          flag++;
}


else if(str[3] == ' ' && str[4] != '(' )
{
          printf("\nOpening parenthesis absent after for keyword");
          flag++;
}


else if(bracket1 != 1 || bracket2 != 1 || bracket1 != bracket2)
{
          printf("\nParentheses Count Error");
          flag++;
}

if(flag == 0)
          printf("\nNo error");
```

```
}

int main(void) {

        char str1[100] = "for (i = 10; i < 20; i++)";
        isCorrect(str1);

        char str2[100] = "for i = 10; i < 20; i++)";
        isCorrect(str2);

        return 0;
}
```

Output:



Output

```
/tmp/TR2YrqnQVf.o

No error
Opening parenthesis absent after for keyword
```

Conclusion:

Correctness Verification:
The code effectively checks various aspects of a for loop statement, such as the presence of the for keyword, correct usage of parentheses, and semicolons.
It provides informative error messages indicating the specific errors found in the input string.

Limitations:

The code assumes that the input strings are representative of for loop statements and may not handle all possible variations or edge cases.
It does not perform semantic analysis or check for logical errors within the loop statement.

Time Complexity:
The time complexity of the code primarily depends on the length of the input string.
The main loop iterates through each character of the string once, resulting in a time complexity of O(n), where n is the length of the input string.

Space Complexity:
The space complexity of the code is constant as it only uses a fixed-size array arr and a few integer variables for counting occurrences.
It does not dynamically allocate memory or use additional data structures based on the input size, resulting in a space complexity of O(1).

Practical 6

Aim:

Write a program to illustrate the generation of SPM for a given grammar.

Theory:

This code is designed to generate a syntax precedence matrix (SPM) based on a given context-free grammar. The grammar is represented as a list of production rules, where each rule consists of a left-hand side (lhs) and a right-hand side (rhs). The code constructs the SPM by analyzing the first and last symbols of the rhs of each production rule and determining their precedence relationships.

Steps in the Code:

Extract Symbols:

Extract all symbols from the grammar, including both lhs and rhs symbols.

Warshall's Algorithm:

Implement Warshall's Algorithm to compute the transitive closure of the "less than" relation between symbols based on the first symbol in the rhs of the production rules.

Compute Last Plus:

Compute the transitive closure of the "greater than" relation between symbols based on the last symbol in the rhs of the production rules.

Compute Equality Set:

Determine the equality relation between symbols based on the pairs of symbols appearing together in the rhs of the production rules.

Compute Precedence Matrix:

Construct the SPM using the computed relations (less than, greater than, and equal).

Print Precedence Matrix:

Display the SPM, showing the precedence relationships between symbols.

Code:

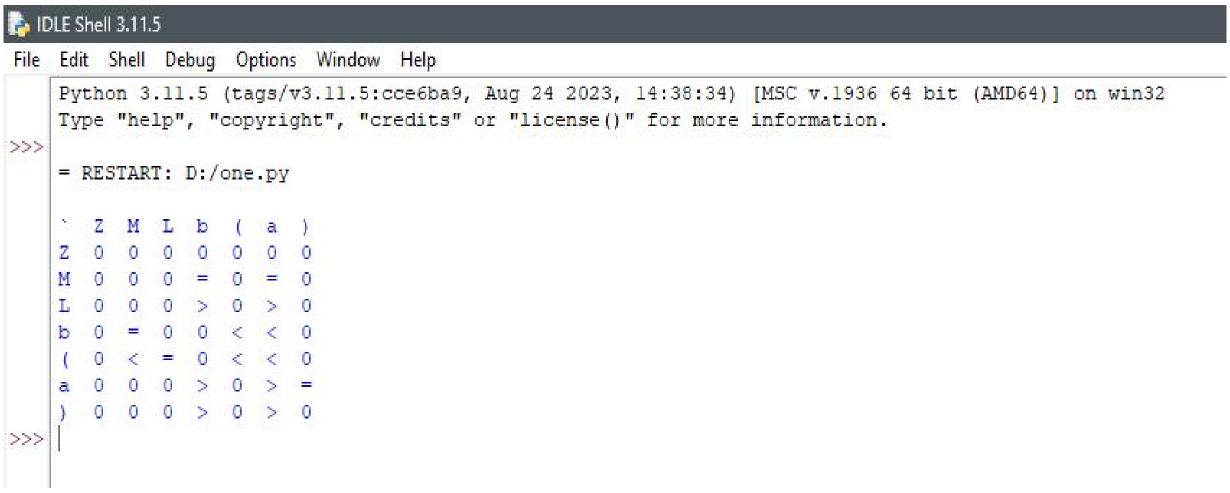
```
grammer = [["Z","bMb"],["M","(L"],['M',"a"],["L","Ma)"]]

lhs = [i[0] for i in grammer]
rhs = [i[1] for i in grammer]

symbol = lhs + rhs
symbols = []
for i in symbol:
   for x in range(0,len(i)):
      if  i[x] not in symbols:
         symbols.append(i[x])

def warshall(a):
   assert (len(row) == len(a) for row in a)
   n = len(a)
   for k in range(n):
      for i in range(n):
         for j in range(n):
            a[i][j] = a[i][j] or (a[i][k] and a[k][j])
   return a


def emptyMat():
   temp= []
   for i in range(0,len(symbols)):
      x = []
      for i in range(0,len(symbols)):
         x.append(0)
      temp.append(x)
   return temp



firstMatrix = emptyMat()
firstStar = emptyMat()

I = []
```

```
identityX=0
for i in range(0,len(symbols)):
   x = []
   for j in range(0,len(symbols)):
      if j == identityX:
         x.append(1)
      else:
         x.append(0)
   identityX += 1
   I.append(x)

i = 0
for j in range(0, len(I)):
      I[i][j] = 1
      i = i+1

for i in range(0,len(lhs)):
   left = lhs[i]
   right = rhs[i]
   right = right[0]
   for i in range(0,len(symbols)):
      if symbols[i] == left:
         findL = i
         break
   for i in range(0,len(symbols)):
      if symbols[i] == right:
         findR = i
         break
   firstMatrix[findL][findR] = 1

firstPlus = warshall(firstMatrix)


lastMatrix = emptyMat()
lastPlus = emptyMat()

for i in range(0,len(rhs)):
   left = lhs[i]
```

```
    right = rhs[i]
    right = right[-1]
    for i in range(0,len(symbols)):
        if symbols[i] == left:
            findL = i
            break
    for i in range(0,len(symbols)):
        if symbols[i] == right:
            findR = i
            break
    lastMatrix[findL][findR] = 1

lastPlus = warshall(lastMatrix)

lastPlusT = emptyMat()

for i in range(len(lastPlus)):
   # iterate through columns
   for j in range(len(lastPlus[0])):
       lastPlusT[j][i] = lastPlus[i][j]

equal = emptyMat()

print("")
eqSet=[]
for i in rhs:
    if len(i) > 1:

        items = -(-len(i)//2)
        x = 0
        y = 1
        for j in range(0,items):
            temp = i[x] + i [y]
            eqSet.append(temp)
            x += 1
            y += 1

for i in eqSet:
    left = i[0]
    right = i[1]

    for j in range(0,len(symbols)):
        if symbols[j] == left:
            findL = j
            break

    for j in range(0,len(symbols)):
        if symbols[j] == right:
            findR = j
            break
    equal[findL][findR] = 1

lessThen = emptyMat()

for i in range(len(equal)):
    for j in range(len(firstPlus[0])):
        for k in range(len(firstPlus)):
            lessThen[i][j] += equal[i][k] * firstPlus[k][j]

for i in range(0,len(firstPlus)):
    for j in range(0,len(firstPlus[0])):
        #print(f"i={i}  j={j}")
        firstStar[i][j] = firstPlus[i][j] or  I[i][j]

greaterThen = emptyMat()
eqSfp = emptyMat()

for i in range(len(equal)):
    for j in range(len(firstStar[0])):
         for k in range(len(firstStar)):
            eqSfp[i][j] += equal[i][k] * firstStar[k][j]

for i in range(len(lastPlusT)):
    for j in range(len(eqSfp[0])):
         for k in range(len(eqSfp)):
            greaterThen[i][j] += lastPlusT[i][k] * eqSfp[k][j]
```

```
spm = []
for i in range(0,len(symbols)+1):
    x = []
    for i in range(0,len(symbols)+1):
        x.append(0)
    spm.append(x)
spm[0][0] = "`"

for i in range(1,len(spm)):
    spm[0][i] = symbols[i-1]
    spm[i][0] = symbols[i-1]

for i in range(1, len(lessThen)+1):
    for j in range(1, len(lessThen)+1):
        if(equal[i-1][j-1]==1):
            spm[i][j] = "="
        elif(lessThen[i-1][j-1]==1):
            spm[i][j] = "<"
        elif(greaterThen[i-1][j-1]==1):
            spm[i][j] = ">"

for i in spm:
    print ('  '.join(map(str, i)))
```

Output:



```
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [MSC v.1936 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: D:/one.py

`  Z  M  L  b  (  a  )
Z  0  0  0  0  0  0  0
M  0  0  0  =  0  =  0
L  0  0  0  >  0  >  0
b  0  =  0  0  <  <  0
(  0  <  =  0  <  <  0
a  0  0  0  >  0  >  =
)  0  0  0  >  0  >  0
>>> 
```

Conclusion:

The code effectively generates a syntax precedence matrix (SPM) from the given context-free grammar.
The SPM helps in analyzing the precedence relationships between symbols, aiding in the parsing of expressions and grammatical constructs.
By visualizing the precedence matrix, developers can identify potential conflicts or ambiguities in the grammar.

Time Complexity:

The time complexity primarily depends on the size of the grammar and the number of symbols. The code utilizes Warshall's Algorithm, which has a time complexity of O(n^3), where n is the number of symbols. Therefore, the overall time complexity can be considered to be O(n^3).

Space Complexity:

The space complexity is determined by the size of the syntax precedence matrix (SPM), which is proportional to the square of the number of symbols. Hence, the space complexity can be expressed as O(n^2), where n is the number of symbols.

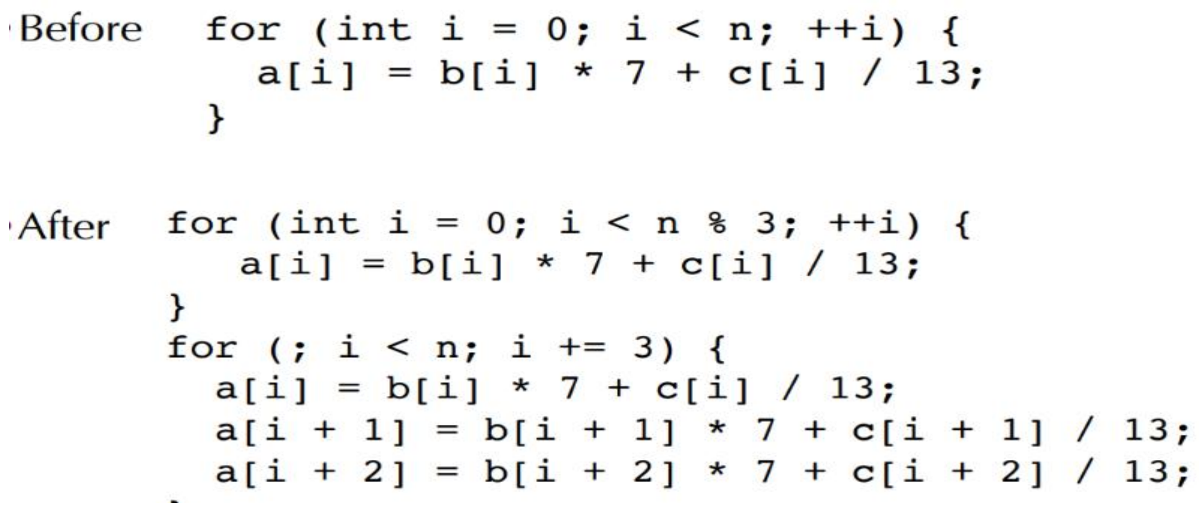
Practical 7

Aim:

Write a program to demonstrate loop unrolling and loop splitting for the given code sequence containing loop.

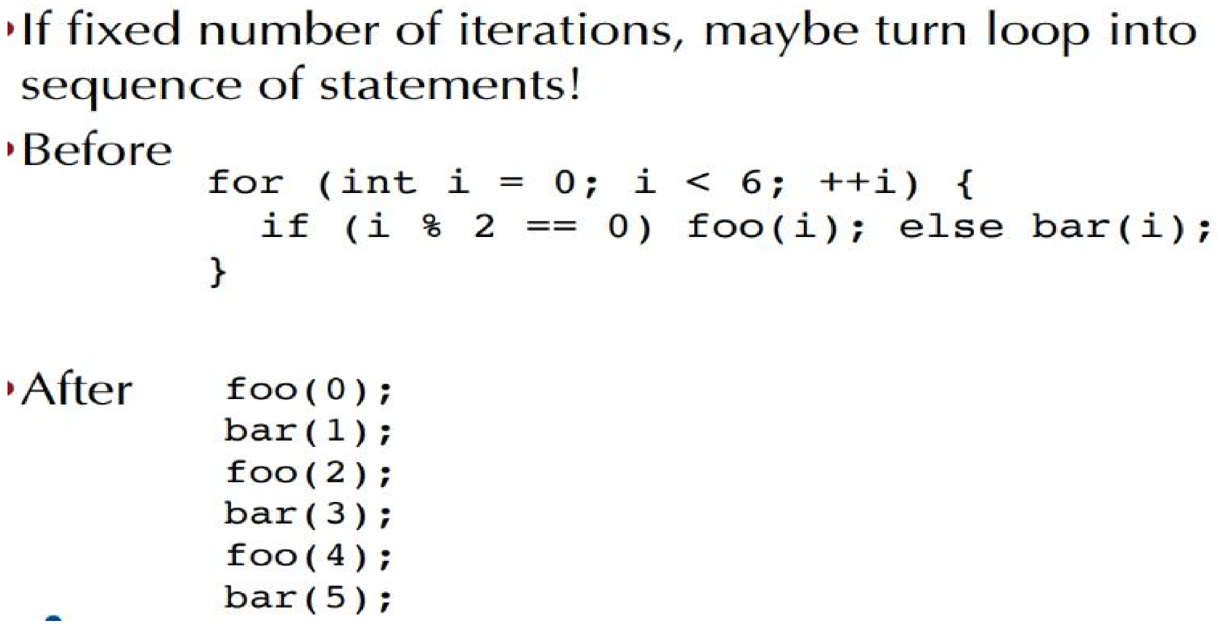
Theory:

Loop unrolling is a loop transformation technique that helps to optimize the execution time of a program. We basically remove or reduce iterations. Loop unrolling increases the program's speed by eliminating loop control instruction and loop test instructions.



Before
```
for (int i = 0; i < n; ++i) {
  a[i] = b[i] * 7 + c[i] / 13;
}
```

After
```
for (int i = 0; i < n % 3; ++i) {
   a[i] = b[i] * 7 + c[i] / 13;
}
for (; i < n; i += 3) {
  a[i] = b[i] * 7 + c[i] / 13;
  a[i + 1] = b[i + 1] * 7 + c[i + 1] / 13;
  a[i + 2] = b[i + 2] * 7 + c[i + 2] / 13;
```



- If fixed number of iterations, maybe turn loop into sequence of statements!
- Before

```
for (int i = 0; i < 6; ++i) {
  if (i % 2 == 0) foo(i); else bar(i);
}
```

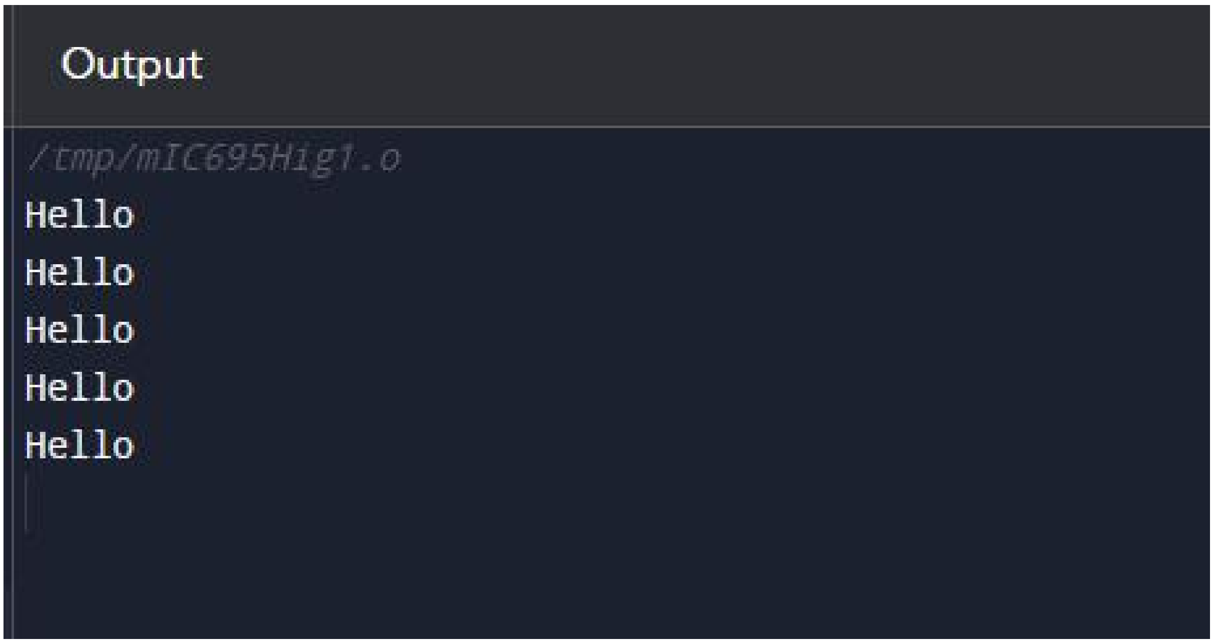
- After

```
foo(0);
bar(1);
foo(2);
bar(3);
foo(4);
bar(5);
```

Code 1 :

```
// This program does not uses loop unrolling.
#include<stdio.h>
int main(void)
{
        for (int i=0; i<5; i++)
                printf("Hello\n"); //print hello 5 times
        return 0;
}
```

Output 1 :



```
Output
/tmp/mIC695Hig1.o
Hello
Hello
Hello
Hello
Hello
```

Conclusion for first code:

The code is concise and demonstrates the use of a for loop in C to repeat a specific task multiple times. It serves as a basic example for beginners to understand loop structures in programming. The program effectively prints "Hello" five times and then exits.

Time Complexity:

The time complexity of this code is O(1), as the loop iterates a fixed number of times (5 iterations) regardless of the input size.

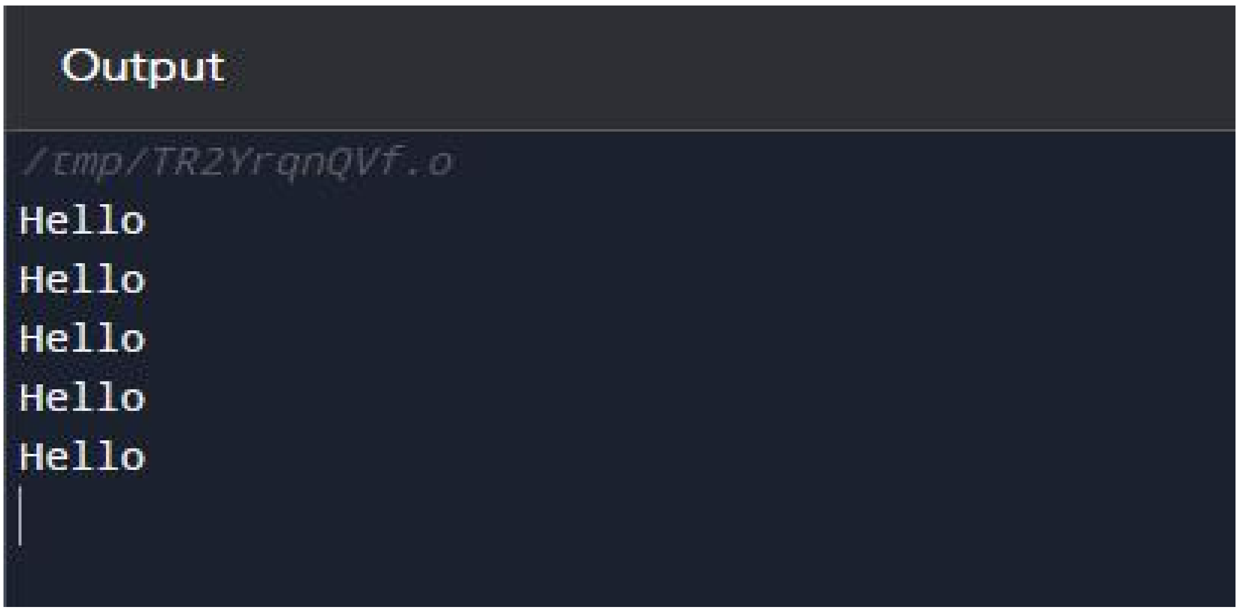
Space Complexity:

The space complexity is O(1), as the program only uses a fixed amount of memory to store the loop variable i and other program instructions, which do not depend on the input size.

Code 2 :

```
// This program uses loop unrolling.
#include<stdio.h>
int main(void)
{
	// unrolled the for loop in program 1
	printf("Hello\n");
	printf("Hello\n");
	printf("Hello\n");
	printf("Hello\n");
	printf("Hello\n");
	return 0;
}
```

Output 2 :



```
Output
/tmp/TR2YrqnQVf.o
Hello
Hello
Hello
Hello
Hello
```

Conclusion for second code:

This code achieves the same result as the previous code but in a more straightforward manner by unrolling the loop.
Loop unrolling can sometimes improve performance by reducing loop overhead, but in this case, it is a micro-optimization with negligible impact.
It provides a clear and concise demonstration of achieving repetition without using a loop.

Time Complexity:

The time complexity of this code is O(1), as each printf statement is executed once sequentially, resulting in constant time complexity.

Space Complexity:

The space complexity is O(1), as the program only uses a fixed amount of memory to store program instructions and variables, which do not depend on the input size.